24 | 活动的描述：到底什么是进程？

2021-07-02 LMOS

《操作系统实战45讲》 课程介绍 >

讲述：陈晨

时长 18:15 大小 16.72M

你好，我是 LMOS。

在前面的课程里，我们已经实现了数据同步、hal 层的初始化，中断框架、物理内存、内存对象、虚拟内存管理，这些都是操作系统中最核心的东西。

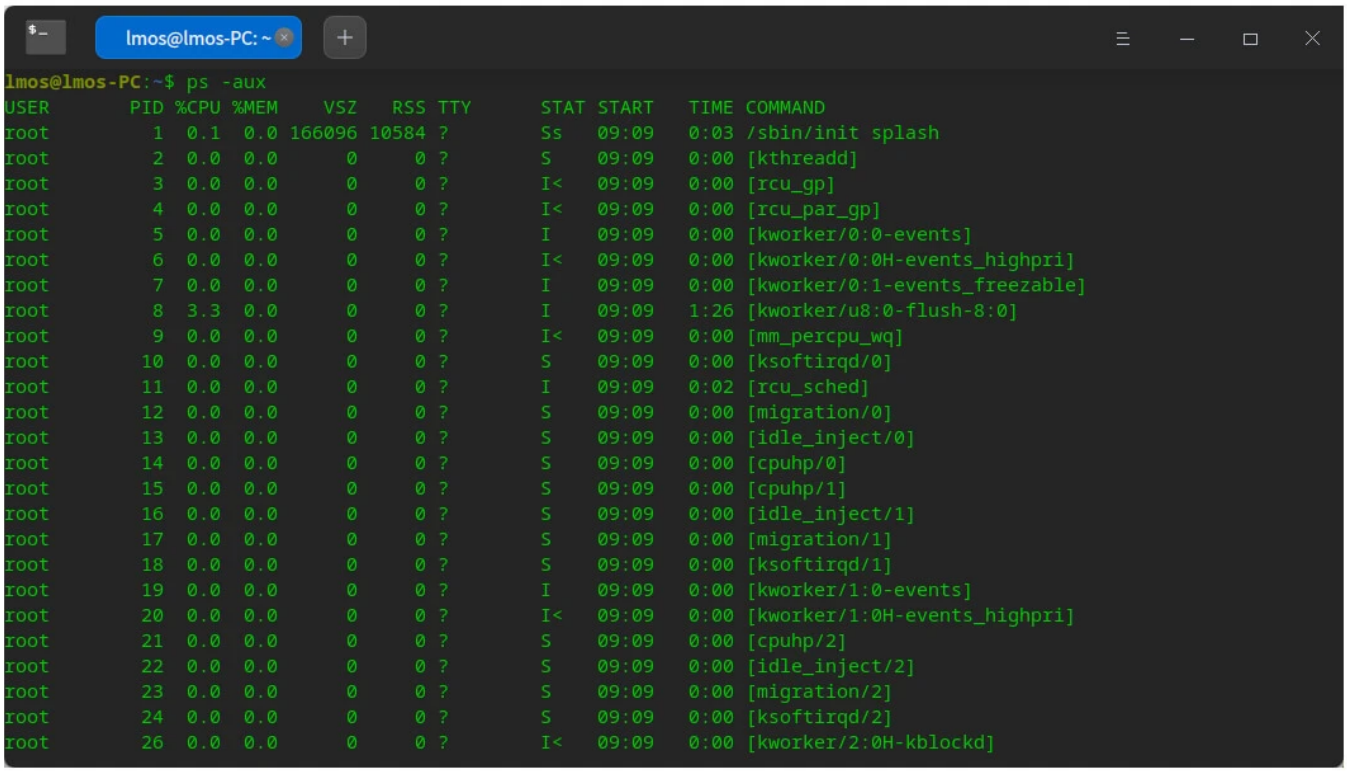
今天，我再给你讲讲操作系统里一个层次非常高的组件——进程 ，而它又非常依赖于内存管理、中断、硬件体系结构。好在前面课程中，这些基础知识我们已经搞得清清楚楚，安排得明明白白了，所以我们今天理解进程就变得顺理成章。

感受一下

在你看来，什么是进程呢？日常我们跟计算机打交道的时候，最常接触的就是一些应用程序，比如 Word、浏览器，你可以直观感受到它们的存在。而我们却很难直观感受到什么是进程，自然也就不容易描述它的模样与形态了。

其实，在我们启用 Word 这些应用时，操作系统在背后就会建立至少一个进程。虽然我们难以观察它的形态，但我们绝对可以通过一些状态数据来发现进程的存在。

在 Linux 的终端下输入 ps 命令， 我们就可以看到系统中有多少个进程了。如下图所示。



这是进程吗？是的，不过这只是一些具体进程的数据，如创建进程和用户、进程 ID、使用 CPU 的百分比，进程运行状态，进程的建立时间、进程的运行时间、进程名等，这些数据综合起来就代表了一个进程。

也许看到这，你会呵呵一笑，觉得原来抽象的进程背后，不过是一堆数据而已，关于进程这就是我们能直观感受到的东西，这就完了吗？当然没有，我们接着往下看。

什么是进程

如果你要组织一个活动怎么办？你首先会想到，这个活动的流程是什么，需要配备哪些人员和物资，中途要不要休息，活动当前进行到哪里了……如果你是个精明的人，你大概会用**表格**把这些信息记录下来。

同理，你运行一个应用程序时，操作系统也要记录这个应用程序使用多少内存，打开了什么文件，当有些资源不可用的时候要不要睡眠，当前进程运行到哪里了。操作系统把这些信息综合统计，存放在内存中，抽象为进程。

现在你就可以回答什么是进程了：进程是一个应用程序运行时刻的实例（从进程的结构看）；进程是应用程序运行时所需资源的容器（从进程的功能看）；甚至进程是一堆数据结构（从操作系统对进程实现的角度来说）。

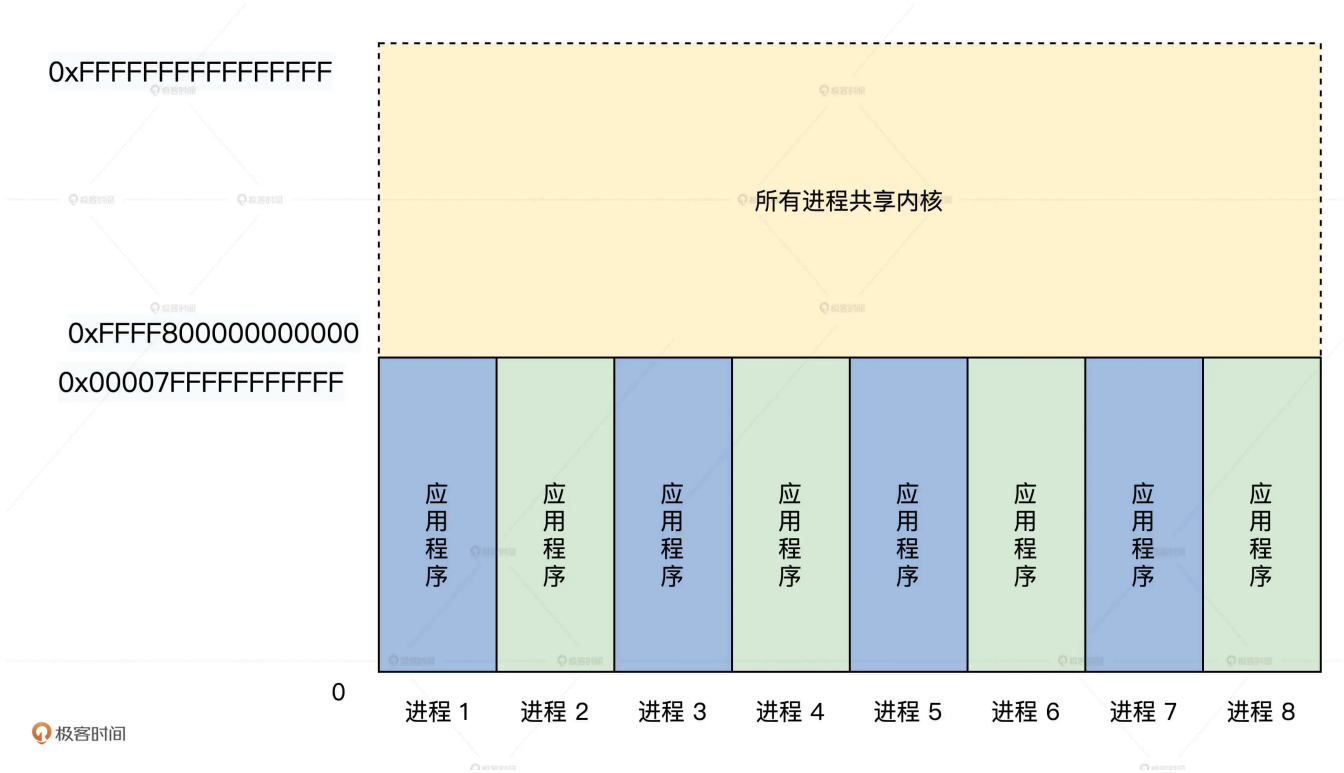
这也太简单了吧？对，进程的抽象概念就是这么简单。我知道这一定不能让你真正明白什么是进程，抽象的概念就是如此，你不在实践中设计并实现它，是很难真正明白的。下面我们先来细化设计。

进程的结构

首先，进程是一个应用程序运行时刻的实例，它的目的就是操作系统用于管理和运行多个应用程序的；其次，从前面我们实现的内存管理组件角度看，操作系统是给应用程序提供服务的。

所以，从这两个角度看，进程必须要有一个地址空间，这个地址空间至少包括两部分内容：一部分是内核，一部分是用户的应用程序。

最后，结合 x86 硬件平台对虚拟地址空间的制约，我给你画了一幅图，如下所示。



进程结构示意图

上图中有 8 个进程，每个进程拥有 x86 CPU 的整个虚拟地址空间，这个虚拟地址空间被分成了两个部分，上半部分是所有进程都共享的内核部分 ，里面放着一份内核代码和数据，下半部分是应用程序，分别独立，互不干扰。

还记得我们讲过的 x86 CPU 的特权级吗？

当 CPU 在 R0 特权级运行时，就运行在上半部分内核的地址空间中，当 CPU 在 R3 特权级时，就运行在下半部分的应用程序地址空间中。各进程的虚拟地址空间是相同的，它们之间物理地址不同，是由 MMU 页表进行隔离的，所以每个进程的应用程序的代码绝对不能随意访问内核的代码和数据。

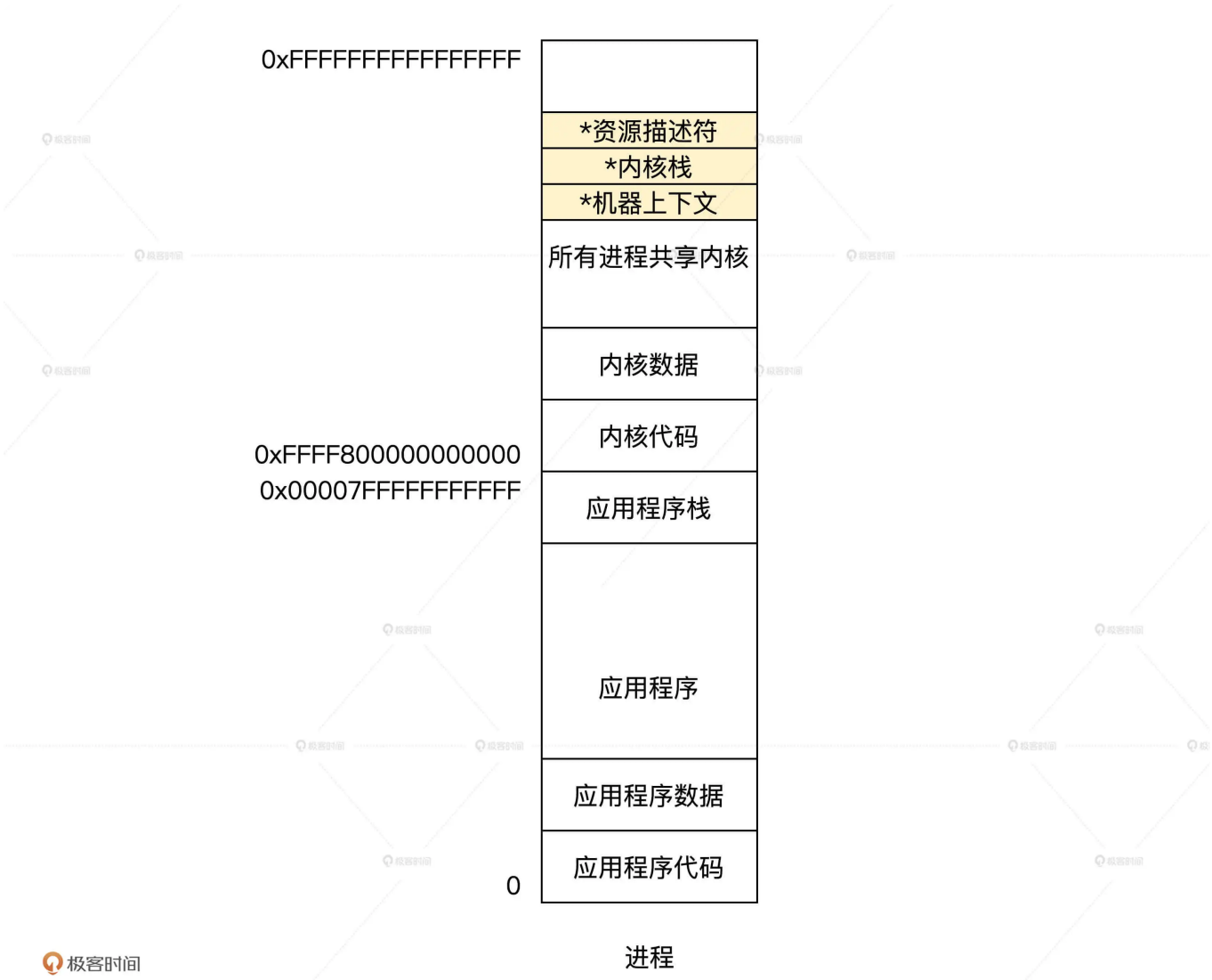
以上是整体结构，下面我们来细化一下进程需要实现哪些功能？

我们先从**应用程序和内核的关系**看。应用程序需要内核提供资源，而内核需要控制应用程序的运行。那么内核必须能够命令应用程序，让它**随时中断（进入内核地址空间）或恢复执行**，这就需要保存应用程序的机器上下文和它运行时刻的栈。

接着，我们深入**内核提供服务的机制**。众所周知，内核是这样提供服务的：通过停止应用程序代码运行，进入内核地址空间运行内核代码，然后返回结果。就像活动组织者会用表

格备案一样，内核还需要记录一个应用程序都访问了哪些资源，比如打开了某个文件，或是访问了某个设备。而这样的“记录表”，我们就用“**资源描述符**”来表示。

而我们前面已经说了，进程是一个应用程序运行时刻的实例。那这样一来，一个细化的进程结构，就可以像下图这样设计。



进程结构细化示意图

上图中表示了一个进程详细且必要的结构，其中带 * 号是每个进程都有独立一份，有了这样的设计结构，多个进程就能并发运行了。前面这些内容还是纸上谈兵，你重点搞明白进程的概念和结构就行了。

实现进程

前面我们简单介绍了进程的概念和结构，之所以简单，是为了不在理论层面就把问题复杂化，这对我们实现 Cosmos 的进程组件没有任何好处。

但只懂理论还是空中阁楼，我们可以一步步在设计实现中，由浅到深地理解什么是进程。我们这就把前面的概念和设计，一步步落实到代码，设计出对应的数据结构。

如何表示一个进程

根据前面课程的经验，如果要在软件代码中表示一个什么东西时，就要设计出对应的数据结构。

那么对于一个进程，它有状态，id，运行时间，优先级，应用程序栈，内核栈，机器上下文，资源描述符，地址空间，我们将这些信息组织在一起，就形成了一个进程的数据结构。

下面我带你把它变成代码，在 cosmos/include/knlinc/ 目录下建立一个 krlthread_t.h 文件，在其中写上代码，如下所示。

复制代码

```
typedef struct s_THREAD
{
    spinlock_t  td_lock;            //进程的自旋锁
    list_h_t    td_list;            //进程链表
    uint_t      td_flgs;            //进程的标志
    uint_t      td_stus;            //进程的状态
    uint_t      td_cpuid;           //进程所在的CPU的id
    uint_t      td_id;              //进程的id
    uint_t      td_tick;            //进程运行了多少tick
    uint_t      td_privilege;       //进程的权限
    uint_t      td_priority;        //进程的优先级
    uint_t      td_runmode;         //进程的运行模式
    adr_t       td_krlstktop;       //应用程序内核栈顶地址
    adr_t       td_krlstkstart;     //应用程序内核栈开始地址
    adr_t       td_usrstktop;       //应用程序栈顶地址
    adr_t       td_usrstkstart;     //应用程序栈开始地址
    mmadrsdsc_t* td_mmdsc;          //地址空间结构
    context_t   td_context;         //机器上下文件结构
    objnode_t*  td_handtbl[TD_HAND_MAX];//打开的对象数组
}thread_t;
```

在 Cosmos 中，我们就使用 thread_t 结构的一个实例变量代表一个进程。进程的内核栈和进程的应用程序栈是两块内存空间，进程的权限表示一个进程是用户进程还是系统进程。进程的权限不同，它们能完成功能也不同。

万事都有轻重缓急，进程也一样，进程有 64 个优先级，td_priority 数值越小优先级越高。td_handtbl 只是一个 objnode_t 结构的指针类型数组。

比方说，一个进程打开一个文件内核就会创建一个对应的 objnode_t 结构的实例变量，这个 objnode_t 结构的地址就保存在 td_handtbl 数组中。你可以这么理解：这个 objnode_t 结构就是进程打开资源的描述符。

进程的地址空间

在 thread_t 结构中有个 mmadrsdsc_t 结构的指针，在这个结构中有虚拟地址区间结构和 MMU 相关的信息。mmadrsdsc_t 结构你应该很熟悉，在虚拟内存那节课中，我们学习过，今天我们再次复习一下，如下所示。

复制代码

```
typedef struct s_MMADRSDSC
{
    spinlock_t msd_lock;                //保护自身的自旋锁
    list_h_t msd_list;                  //链表
    uint_t msd_flag;                    //状态和标志
    uint_t msd_stus;
    uint_t msd_scount;                  //计数，该结构可能被共享
    sem_t  msd_sem;                     //信号量
    mmudsc_t msd_mmu;                   //MMU页表相关的信息
    virmemadrs_t msd_virmemadrs;        //虚拟地址空间结构
    adr_t msd_stext;                    //应用的指令区的开始、结束地址
    adr_t msd_etext;
    adr_t msd_sdata;                    //应用的数据区的开始、结束地址
    adr_t msd_edata;
    adr_t msd_sbss;                     //应用初始化为0的区域开始、结束地址
    adr_t msd_ebss;
    adr_t msd_sbrk;                     //应用的堆区的开始、结束地址
    adr_t msd_ebrk;
}mmadrsdsc_t;
```

上述代码中，注释已经很清楚了，mmadrsdsc_t 结构描述了一个进程的完整的地址空间。需要搞清楚的是：在常规情况下，新建一个进程就要建立一个 mmadrsdsc_t 结构，让 thread_t 结构的 td_mmdsc 的指针变量指向它。

进程的机器上下文

进程的机器上下文分为几个部分，一部分是 CPU 寄存器，一部分是内核函数调用路径。CPU 的通用寄存器，是中断发生进入内核时，压入内核栈中的，从中断入口处开始调用的函数，都是属于内核的函数。

函数的调用路径就在内核栈中，整个过程是这样的：进程调度器函数会调用进程切换函数，完成切换进程这个操作，而**在进程切换函数中会保存栈寄存器的值。**好，下面我们来设计这样一个结构来保存这些信息。

复制代码

```
typedef struct s_CONTEXT
{
    uint_t       ctx_nextrip; //保存下一次运行的地址
    uint_t       ctx_nextrsp; //保存下一次运行时内核栈的地址
    x64tss_t*    ctx_nexttss; //指向tss结构
}context_t;
```

context_t 结构中的字段不多，我们相对陌生的就是 x64tss_t 结构的指针，这个结构是 CPU 要求的一个结构，这个结构它本身的地址放在一个 GDT 表项中，由 CPU 的 tr 寄存器指向，tr 寄存器中的值是 GDT 中 x64tss_t 结构项对应的索引。x64tss_t 结构的代码如下所示。

复制代码

```
// cosmos/hal/x86/halglobal.c
// 每个CPU核心一个tss
HAL_DEFGLOB_VARIABLE(x64tss_t,x64tss)[CPUCORE_MAX];

typedef struct s_X64TSS
{
    u32_t reserv0; //保留
    u64_t rsp0;  //R0特权级的栈地址
    u64_t rsp1;  //R1特权级的栈地址，我们未使用
    u64_t rsp2;  //R2特权级的栈地址，我们未使用
    u64_t reserv28;//保留
    u64_t ist[7];  //我们未使用
    u64_t reserv92;//保留
    u16_t reserv100;//保留
    u16_t iobase;   //我们未使用
}__attribute__((packed)) x64tss_t;
```

CPU 在发生中断时，会根据中断门描述里的目标段选择子，进行必要的特权级切换，特权级的切换就必须要切换栈，CPU 硬件会自己把当前 rsp 寄存器保存到内部的临时寄存器 tmprsp；然后从 x64tss_t 结构体中找出对应的栈地址，装入 rsp 寄存器中；接着，再把当前的 ss、tmprsp、rflags、cs、rip，依次压入当前 rsp 指向的栈中。

建立进程

之前我们已经设计好了进程相关的数据结构，现在我们要讨论如何建立一个新的进程了。建立进程非常简单，就是在内存中建立起对应的数据结构的实例变量。

但是对进程来说，并不是建立 thread_t 结构的实例变量就完事了，还要建立进程的应用程序栈和进程的内核栈，进程地址空间等。下面我们一起来实现建立进程的功能。

建立进程接口

我们先从建立进程的接口开始写起，先在 cosmos/kernel/ 目录下新建一个文件 krlthread.c，在其中写上一个函数。接口函数总是简单的，代码如下所示。

复制代码

```
thread_t *krlnew_thread(void *filerun, uint_t flg, uint_t prilg, uint_t prity,
{
    size_t tustksz = usrstksz, tkstksz = krlstksz;
    //对参数进行检查，不合乎要求就返回NULL表示创建失败
    if (filerun == NULL || usrstksz > DAFT_TDUSRSTKSZ || krlstksz > DAFT_TDKRL
    {
        return NULL;
    }
    if ((prilg != PRILG_USR && prilg != PRILG_SYS) || (prity >= PRITY_MAX))
    {
        return NULL;
    }
    //进程应用程序栈大小检查，大于默认大小则使用默认大小
    if (usrstksz < DAFT_TDUSRSTKSZ)
    {
        tustksz = DAFT_TDUSRSTKSZ;
    }
    //进程内核栈大小检查，大于默认大小则使用默认大小
    if (krlstksz < DAFT_TDKRLSTKSZ)
    {
        tkstksz = DAFT_TDKRLSTKSZ;
    }
    //是否建立内核进程
    if (KERNTHREAD_FLG == flg)
    {
```

```
26          return krlnew_kern_thread_core(filerun, flg, prilg, prity, tustksz, tk
27      }
28      //是否建立普通进程
29      else if (USERTHREAD_FLG == flg)
30      {
31          return krlnew_user_thread_core(filerun, flg, prilg, prity, tustksz, tk
32      }
33      return NULL;
34  }
```

上述代码中的 krlnew_thread 函数的流程非常简单，对参数进行合理检查，其参数从左到右分别是应用程序启动运行的地址、创建标志、进程权限和进程优先级、进程的应用程序栈和内核栈大小。

进程对栈的大小有要求，如果小于默认大小 8KB 就使用默认的栈大小，最后根据创建标志确认是建立内核态进程还是建立普通进程。

建立内核进程

什么是内核进程？你一定在想，其实内核进程就是**用进程的方式去运行一段内核代码，那么这段代码就可以随时暂停或者继续运行，又或者和其它代码段并发运行，只是这种进程永远不会回到进程应用程序地址空间中去，只会在内核地址空间中运行。**

下面我来写代码实现建立一个内核态进程，如下所示。

复制代码

```
1  thread_t *krlnew_kern_thread_core(void *filerun, uint_t flg, uint_t prilg, uin
2  {
3      thread_t *ret_td = NULL;
4      bool_t acs = FALSE;
5      adr_t krlstkadr = NULL;
6      //分配内核栈空间
7      krlstkadr = krlnew(krlstksz);
8      if (krlstkadr == NULL)
9      {
10         return NULL;
11     }
12     //建立thread_t结构体的实例变量
13     ret_td = krlnew_thread_dsc();
14     if (ret_td == NULL)
15     {//创建失败必须要释放之前的栈空间
16         acs = krldelete(krlstkadr, krlstksz);
17         if (acs == FALSE)
```

```
        {
            return NULL;
        }
        return NULL;
    }
    //设置进程权限
    ret_td->td_privilege = prilg;
    //设置进程优先级
    ret_td->td_priority = prity;
    //设置进程的内核栈顶和内核栈开始地址
    ret_td->td_krlstktop = krlstkadr + (adr_t)(krlstksz - 1);
    ret_td->td_krlstkstart = krlstkadr;
    //初始化进程的内核栈
    krlthread_kernstack_init(ret_td, filerun, KMOD_EFLAGS);
    //加入进程调度系统
    krlschdclass_add_thread(ret_td);
    //返回进程指针
    return ret_td;
}
```

上述代码的逻辑非常简单，首先分配一个内核栈的内存空间，接着创建 thread_t 结构的实例变量，然后对 thtead_t 结构体的字段进行设置，最后，初始化进程内核栈把这个新进程加入到进程的调度系统之中，下面来一步步写入实现这些逻辑的代码。

创建 thread_t 结构

创建 thread_t 结构，其实就是分配一块内存用于存放 thread_t 结构的实例变量。类似这样的操作我们课程里做过多次，相信现在你已经能驾轻就熟了。下面我们来写代码实现这个操作，如下所示。

复制代码

```
//初始化context_t结构
void context_t_init(context_t *initp)
{
    initp->ctx_nextrip = 0;
    initp->ctx_nextrsp = 0;
    //指向当前CPU的tss
    initp->ctx_nexttss = &x64tss[hal_retn_cpuid()];
    return;
}
//返回进程id其实就thread_t结构的地址
uint_t krlretn_thread_id(thread_t *tdp)
{
    return (uint_t)tdp;
}
//初始化thread_t结构
```

```
16 void thread_t_init(thread_t *initp)
17 {
18     krlspinlock_init(&initp->td_lock);
19     list_init(&initp->td_list);
20     initp->td_flgs = TDFLAG_FREE;
21     initp->td_stus = TDSTUS_NEW;//进程状态为新建
22     initp->td_cpuid = hal_retn_cpuid();
23     initp->td_id = krlretn_thread_id(initp);
24     initp->td_tick = 0;
25     initp->td_privilege = PRILG_USR;//普通进程权限
26     initp->td_priority = PRITY_MIN;//最高优先级
27     initp->td_runmode = 0;
28     initp->td_krlstktop = NULL;
29     initp->td_krlstkstart = NULL;
30     initp->td_usrstktop = NULL;
31     initp->td_usrstkstart = NULL;
32     initp->td_mmdsc = &initmmadrsdsc;//指向默认的地址空间结构
33 
34     context_t_init(&initp->td_context);
35     //初始化td_handtbl数组
36     for (uint_t hand = 0; hand < TD_HAND_MAX; hand++)
37     {
38         initp->td_handtbl[hand] = NULL;
39     }
40     return;
41 }
42 //创建thread_t结构
43 thread_t *krlnew_thread_dsc()
44 {
45     //分配thread_t结构大小的内存空间
46     thread_t *rettdp = (thread_t *)(krlnew((size_t)(sizeof(thread_t))));
47     if (rettdp == NULL)
48     {
49         return NULL;
50     }
51     //初始化刚刚分配的thread_t结构
52     thread_t_init(rettdp);
53     return rettdp;
54 }
```

相信凭你现在的能力，上述代码一定是超级简单的。不过我们依然要注意这样几点。

首先，我们以 thread_t 结构的地址作为进程的 ID，这个 ID 具有唯一性；其次，我们目前没有为一个进程分配 mmadrsdsc_t 结构体，而是指向了默认的地址空间结构 initmmadrsdsc；最后，hal_retn_cpuid 函数在目前的情况下永远返回 0，这是因为我们使用了一个 CPU。

初始化内核栈

为什么要初始化进程的内核栈呢？

你也许会想，进程的内核栈无非是一块内存，其实只要初始化为 0 就好。当然不是这么简单，我们初始化进程的内核栈，其实是为了在进程的内核栈中放置一份 CPU 的寄存器数据。

这份 CPU 寄存器数据是一个进程机器上下文的一部分，当一个进程开始运行时，我们将会使用“pop”指令从进程的内核栈中弹出到 CPU 中，这样 CPU 就开始运行进程了，CPU 的一些寄存器是有位置关系的，所以我们要定义一个结构体来操作它们，如下所示。

复制代码

```
typedef struct s_INTSTKREGS
{
    uint_t r_gs;
    uint_t r_fs;
    uint_t r_es;
    uint_t r_ds;  //段寄存器
    uint_t r_r15;
    uint_t r_r14;
    uint_t r_r13;
    uint_t r_r12;
    uint_t r_r11;
    uint_t r_r10;
    uint_t r_r9;
    uint_t r_r8;
    uint_t r_rdi;
    uint_t r_rsi;
    uint_t r_rbp;
    uint_t r_rdx; //通用寄存器
    uint_t r_rcx;
    uint_t r_rbx;
    uint_t r_rax;
    uint_t r_rip_old;//程序的指针寄存器
    uint_t r_cs_old;//代码段寄存器
    uint_t r_rflgs; //rflags标志寄存
    uint_t r_rsp_old;//栈指针寄存器
    uint_t r_ss_old; //栈段寄存器
}intstkregs_t;
```

intstkregs_t 结构中，每个字段都是 8 字节 64 位的，因为 x86 CPU 在长模式下 rsp 栈指针寄存器始终 8 字节对齐。栈是向下伸长的（从高地址向低地址）所以这个结构是反向定

义（相对于栈）如果你不理解这个寄存器位置，可以回到中断处理那节课复习一下。

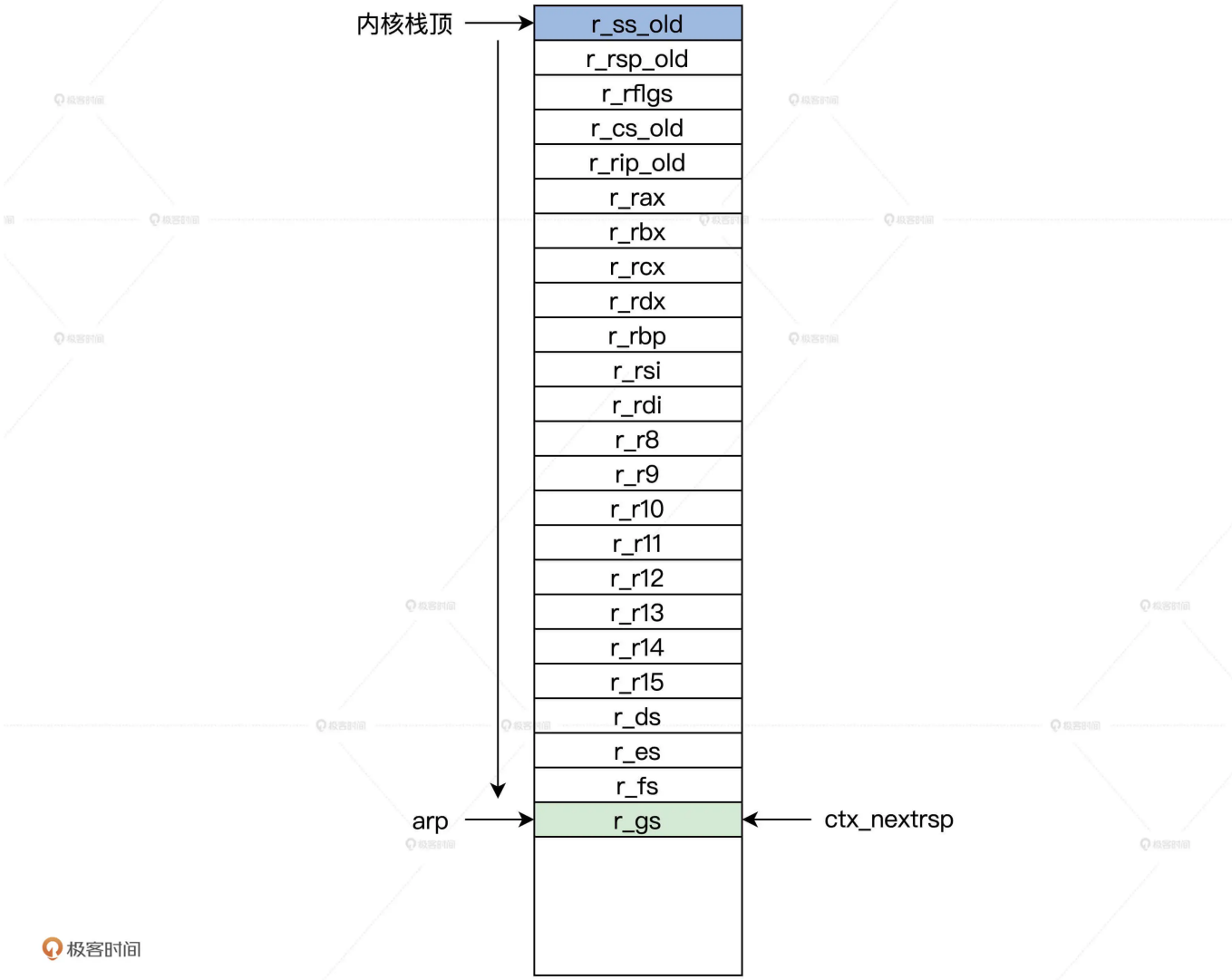
intstkregs_t 结构已经定义好了，下面我们来写代码初始化内核栈，如下所示。

复制代码

```
1  void krlthread_kernstack_init(thread_t *thdp, void *runadr, uint_t cpuflags)
2  {
3      //处理栈顶16字节对齐
4      thdp->td_krlstktop &= (~0xf);
5      thdp->td_usrstktop &= (~0xf);
6      //内核栈顶减去intstkregs_t结构的大小
7      intstkregs_t *arp = (intstkregs_t *)(thdp->td_krlstktop - sizeof(intstkreg
8      //把intstkregs_t结构的空间初始化为0
9      hal_memset((void*)arp, 0, sizeof(intstkregs_t));
10     //rip寄存器的值设为程序运行首地址
11     arp->r_rip_old = (uint_t)runadr;
12     //cs寄存器的值设为内核代码段选择子
13     arp->r_cs_old = K_CS_IDX;
14     arp->r_rflgs = cpuflags;
15     //返回进程的内核栈
16     arp->r_rsp_old = thdp->td_krlstktop;
17     arp->r_ss_old = 0;
18     //其它段寄存器的值设为内核数据段选择子
19     arp->r_ds = K_DS_IDX;
20     arp->r_es = K_DS_IDX;
21     arp->r_fs = K_DS_IDX;
22     arp->r_gs = K_DS_IDX;
23     //设置进程下一次运行的地址为runadr
24     thdp->td_context.ctx_nextrip = (uint_t)runadr;
25     //设置进程下一次运行的栈地址为arp
26     thdp->td_context.ctx_nextrsp = (uint_t)arp;
27     return;
28 }
```

上述代码没什么难点，就是第 7 行我要给你解释一下，arp 为什么要用内核栈顶地址减去 intstkregs_t 结构的大小呢？

C 语言处理结构体时，从结构体第一个字段到最后一个字段，这些字段的地址是从下向上（地址从到高）伸长的，而栈正好相反，所以要减去 intstkregs_t 结构的大小，为 intstkregs_t 结构腾出空间，如下图所示。



内核态进程结构

因为我们建立的是内核态进程，所以上面初始化的内核栈是不能返回到进程的应用程序空间的。而如果要返回到进程的应用程序空间中，内核栈中的内容是不同的，但是内核栈结构却一样。

下面我们动手写代码，初始化返回进程应用程序空间的内核栈。请注意，初始化的还是内核栈，只是内容不同，代码如下所示。

复制代码

```
1 void krlthread_userstack_init(thread_t *thdp, void *runadr, uint_t cpuflags)
2 {
3     //处理栈顶16字节对齐
4     thdp->td_krlstktop &= (~0xf);
5     thdp->td_usrstktop &= (~0xf);
6     //内核栈顶减去intstkregs_t结构的大小
7     intstkregs_t *arp = (intstkregs_t *)(thdp->td_krlstktop - sizeof(intstkreg
8     //把intstkregs_t结构的空间初始化为0
```

```
    hal_memset((void*)arp, 0, sizeof(intstkregs_t));
    //rip寄存器的值设为程序运行首地址
    arp->r_rip_old = (uint_t)runadr;
    //cs寄存器的值设为应用程序代码段选择子
    arp->r_cs_old = U_CS_IDX;
    arp->r_rflgs = cpuflags;
    //返回进程应用程序空间的栈
    arp->r_rsp_old = thdp->td_usrstktop;
    //其它段寄存器的值设为应用程序数据段选择子
    arp->r_ss_old = U_DS_IDX;
    arp->r_ds = U_DS_IDX;
    arp->r_es = U_DS_IDX;
    arp->r_fs = U_DS_IDX;
    arp->r_gs = U_DS_IDX;
    //设置进程下一次运行的地址为runadr
    thdp->td_context.ctx_nextrip = (uint_t)runadr;
    //设置进程下一次运行的栈地址为arp
    thdp->td_context.ctx_nextrsp = (uint_t)arp;
    return;
}
```

上述代码中初始化进程的内核栈，所使用的段选择子指向的是应用程序的代码段和数据段，这个代码段和数据段它们特权级为 R3，CPU 正是根据这个代码段、数据段选择子来切换 CPU 工作特权级的。这样，CPU 的执行流就可以返回到进程的应用程序空间了。

建立普通进程

在建立进程的接口函数 krlnew_thread 的流程中，会根据参数 flg 的值，选择调用不同的函数，来建立不同类型的进程。

前面我们已经写好了建立内核进程的函数，接下来我们还要写好建立普通进程的函数，如下所示。

复制代码

```
thread_t *krlnew_user_thread_core(void *filerun, uint_t flg, uint_t prilg, uin
{
    thread_t *ret_td = NULL;
    bool_t acs = FALSE;
    adr_t usrstkadr = NULL, krlstkadr = NULL;
    //分配应用程序栈空间
    usrstkadr = krlnew(usrstksz);
    if (usrstkadr == NULL)
    {
        return NULL;
    }
```

```
    //分配内核栈空间
    krlstkadr = krlnew(krlstksz);
    if (krlstkadr == NULL)
    {
        if (krldelete(usrstkadr, usrstksz) == FALSE)
        {
            return NULL;
        }
        return NULL;
    }
    //建立thread_t结构体的实例变量
    ret_td = krlnew_thread_dsc();
    //创建失败必须要释放之前的栈空间
    if (ret_td == NULL)
    {
        acs = krldelete(usrstkadr, usrstksz);
        acs = krldelete(krlstkadr, krlstksz);
        if (acs == FALSE)
        {
            return NULL;
        }
        return NULL;
    }
    //设置进程权限
    ret_td->td_privilege = prilg;
    //设置进程优先级
    ret_td->td_priority = prity;
    //设置进程的内核栈顶和内核栈开始地址
    ret_td->td_krlstktop = krlstkadr + (adr_t)(krlstksz - 1);
    ret_td->td_krlstkstart = krlstkadr;
    //设置进程的应用程序栈顶和内核应用程序栈开始地址
    ret_td->td_usrstktop = usrstkadr + (adr_t)(usrstksz - 1);
    ret_td->td_usrstkstart = usrstkadr;
    //初始化返回进程应用程序空间的内核栈
    krlthread_userstack_init(ret_td, filerun, UMOD_EFLAGS);
    //加入调度器系统
    krlschdclass_add_thread(ret_td);
    return ret_td;
}
```

和建立内核进程相比，建立普通进程有两点不同。第一，**多分配了一个应用程序栈**。因为内核进程不会返回到进程的应用程序空间，所以不需要应用程序栈，而普通进程则需要；第二，在最后调用的是 **krlthread_userstack_init 函数**，该函数初始化返回进程应用程序空间的内核栈，这在前面已经介绍过了。

到此为止，我们建立进程的功能已经实现了。但是最后将进程加入到调度系统的函数，我们还没有写，这个函数是进程调度器模块的函数，我们下节课再讨论。

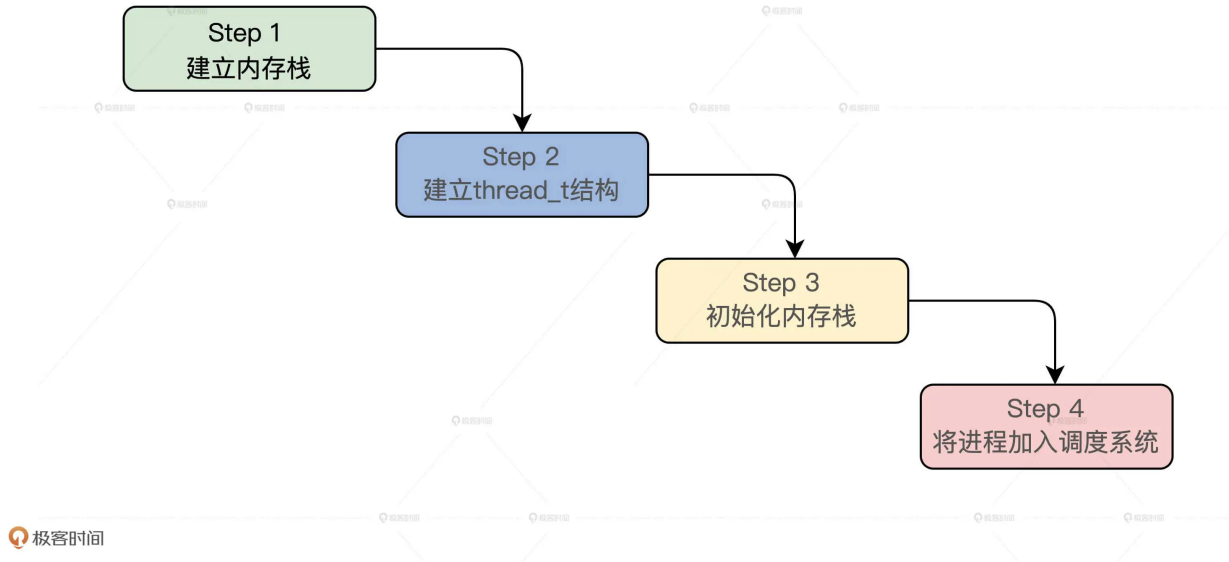
重点回顾

这节课我们用最简洁的方式了解了进程以及如何建立一个进程，我来为你梳理一下今天的课程重点。

首先，我们在 Linux 系统上，用 ps 命令列出 Linux 系统上所有的进程，直观的感受了一下什么进程，从理论上了解了一下进程的结构。

然后我们把进程相关的信息，做了归纳整理，设计出一系列相应的数据结构，这其中包含了表示进程的数据结构，与进程相关内存地址空间结构，还有进程的机器上下文数据结构。这些数据结构综合起来就表示了进程。

最后进入建立进程的环节。有了进程相关的数据结构就可以写代码建立一个进程了，我们的建立进程的接口函数，既能建立普通进程又能建立内核进程，而建立进程的过程无非是创建进程结构体、分配进程的内核栈与应用程序栈，并对进程的内核栈进行初始化，最后将进程加入调度系统，以便后面将进程投入运行。



进程建立流程图

很多理论书籍总是在开头就花大量篇幅讲进程，但你却很难搞懂，这是为什么呢？第一，他们在用抽象方法讲解抽象概念，对初学者很不友好；第二，讲解顺序不对，想搞懂进程，需要前置知识，它是一个高层次的组件。

相信经过前面章节的学习，你现在理解进程会轻松自如。

思考题

请问，各个进程是如何共享同一份内核代码和数据的？

欢迎你在留言区和我交流，相信通过积极参与，你将更好地理解这节课的内容。也欢迎你把这节课分享给你的朋友，说不定可以帮他真正弄懂什么是进程。

好，我是 LMOS，我们下节课见！

分享给需要的人，Ta订阅后你可得 20 元现金奖励

赞 1 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 23 | 瞧一瞧Linux：SLAB如何分配内存？

下一篇 25 | 多个活动要安排（上）：多进程如何调度？

精选留言 (6)

写留言

neohope 置顶
2021-07-06

一、数据结构
thread_t表示一个线程，包括：
锁【并发】、链表
进程标志、进程状态、进程编号【内存地址】、CPU编号【当前全是0】、CPU时间片、进程权限【内核or用户】、优先级【越小越高】、运行模式...
展开

作者回复: 是正确的

1

pedro
2021-07-02

每个进程都有一个内核栈，指向同一个块内核内存区域，共享一份内核代码和内核数据。内核进程一份页表，用户进程两份页表，用户进程多了一份用户空间页表，与其它用户进程互不干扰。

作者回复: 聪明如你

4

blentle
2021-07-02

内存管理算是讲完了吗
展开

编辑回复: 是啊，告一段落了。关键之处课里都讲了，你还可以结合源代码（见Gitee）继续深入学习。

1

嗣树
2021-07-02

各个进程内核空间都使用同一份页表，通过页表映射到同一物理内存。

咱们的进度还是挺快的，一不留神这周都挺近进程了哈哈哈

作者回复: 是的

1

青玉白露
2021-07-07

请问，各个进程是如何共享同一份内核代码和数据的？
——描述进程的结构体中已经写明了：
adr_t td_krlstktop; //应用程序内核栈顶地址
adr_t td_krlstkstart; //应用程序内核栈开始地址
adr_t td_usrstktop; //应用程序栈顶地址 ...
展开

作者回复: 不对哦 你再想想

1

springXu
2021-07-02

这课的问题，让我想起了第20讲留下的小坑。原文如下

进程的物理地址空间，其实可以用一组 MMU 的页表数据表示，它保存在 mmudsc_t 数据结构中，但是这个数据结构我们不在这里研究，放在后面再研究。
...
展开

作者回复: 哈哈 我留坑了？

1